ITRONデバッギングインタフェース仕様における標準化アプローチとその適応性に関する評価

若林 隆行† 高田 広章†

組込みシステム用のデバッグ環境にOSサポート機能を実装するには，OSの内部構造に依存する部分が避けられない．μITRON仕様は，OSのAPIのみを標準化しており，それに準拠して実装されたOSの内部構造はそれぞれ異なることが，デバッグ環境がμITRON仕様OSをサポートする際の障害となっている．この問題を解決するために，我々は，デバッグ環境がμITRON仕様OSをサポートするための標準インタフェースであるITRONデバッギングインタフェース仕様の検討を行っている．ITRONデバッギングインタフェース仕様では，多様なターゲットシステムへの適応性を重視して，OSの内部構造の違いを吸収するモジュールをホスト側に載せるという標準化アプローチをとった．本論文では，ITRONデバッギングインタフェース仕様における標準化アプローチについて述べ，その適応性を評価する．

Standardization Approach of ITRON Debugging Interface Specification and Evaluations of Its Adaptability

Takayuki Wakabayashi† and Hiroaki Takada†

In order to implement OS support functions, debugging environments for embedded systems unavoidably depend on the internal structure of the OS. Because the μITRON specification standardizes only the API, the internal structure of each OS conforming to the μITRON specification is different, resulting in a difficulty for debugging environments to support μITRON-specification OSs. To solve this problem, we are designing the ITRON debugging interface specification which is to define a standard interface for debugging environments to support μITRON-specification OSs. In designing the ITRON debugging interface, we adopted the approach with which a module hiding the difference of the internal structure of OS is embedded to the host side, in order to achieve the adaptability to a variety of target systems. This paper describes the standardization approach adopted to the ITRON debugging interface and evaluates its adaptability.

1. はじめに

μITRON仕様[1)]は，組込みシステム分野で広く用いられているリアルタイムOS（以下，RTOS）仕様である[2)]．異なるプロセッサや応用分野を対象として，数多くのμITRON仕様に準拠したRTOS（以下，μITRON仕様OS）が開発されているが，μITRON仕様はRTOSのAPIのみを標準化しており，それぞれのμITRON仕様OSの内部構造は，ターゲットとするプロセッサや応用分野に最適になるように設計されている．

組込みシステムのソフトウェアのデバッグ作業には，ソフトウェアによるデバッガやインサーキットエミュレータ（以下，ICE）など，様々なデバッグ環境が用いられる．RTOS上で動作するアプリケーションのデバッグ作業を効率的に行うために，デバッグ環境には，OSの管理するオブジェクトの状態表示や操作，タスク実行履歴の取得や表示など，OSに依存したデバッグ支援機能を持つことが求められる．以下では，OSに依存するこれらのデバッグ支援機能を，デバッグ環境のOSサポート機能と呼ぶ．また，デバッグ環境がこれらの機能を持つことを，デバッグ環境がOSをサポートするという．

組込みシステム用のデバッグ環境にμITRON仕様OSをサポートさせようとした場合に，μITRON仕様OSの内部構造がそれぞれ異なっていることが，次のような困難をもたらしている．デバッグ環境に，μITRON仕様OSなどの組込みシステム用のRTOSをサポートさせる場合，デバッグ環境の実装がOSの内部構造に

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

依存することは避けられない☆．そのため，デバッグ環境に1種類のμITRON仕様OSをサポートする機能を実装しても，他のμITRON仕様OSをサポートできることにはならず，数多くのμITRON仕様OSをサポートするためには大きな開発コストがかかることになる．実際，既存のμITRON仕様OSをサポートするデバッグ環境はいずれも，限られた種類のμITRON仕様OSのみを対象としている[3)]．

この問題は，同一のメーカがμITRON仕様OSとデバッグ環境の両方を提供している場合にはそれほど問題にはならないが，μITRON仕様OSの場合，半導体メーカが自社のプロセッサ用にOSを提供し，独立したツールメーカがデバッグ環境を提供しているケースが多い☆☆．また，アプリケーション開発者が，限られた組合せのμITRON仕様OSとデバッグ環境のみを用いるのであれば問題は小さいが，実際には，アプリケーションの要求に最も適合したプロセッサや，それをターゲットしたμITRON仕様OSを選択することが多い．その結果，それまで用いてきたデバッグ環境が利用できなくなるとすれば，アプリケーション開発者にとっては大きい問題である．

この問題に対し，我々は，デバッグ環境がμITRON仕様OSをサポートするための標準インタフェース仕様の検討を行っている．このインタフェース仕様をITRONデバッギングインタフェース仕様と呼び，現時点までにその暫定仕様書を完成させている[4)]．

ITRONデバッギングインタフェース仕様の標準化においては，デバッグを支援するためにOSが持つべき機能を標準化するのではなく，OSの内部構造の違いを吸収するモジュールをホスト側に載せるという標準化アプローチをとった．いい換えると，μITRON仕様OSをサポートするためにデバッグ環境が持つべき機能の中で，OSの内部構造に依存する部分をモジュールとして切り出し，そのモジュールをOSメーカが提供することで，デバッグツール本体はOSの内部構造に依存せずに実装できるようにする．この標準化アプローチにより，ターゲットシステムの特性にあわせてOSサポート機能の実装方法を変え，多様なシステム環境に適応することが可能になる．これを，ターゲットシステムへの適応性と呼ぶ．

☆ たとえば，OSオブジェクトの状態を読み出すシステムコールが用意されていれば，OSの内部構造に依存せずにOSオブジェクトの状態表示を実現することができるが，μITRON仕様OSなどの組込みシステム用のRTOSは，そのようなシステムコールを持っているとは限らない．

☆☆ μITRON仕様OSを提供している独立したOSメーカもあるが，ツールメーカとは一致していないケースが多い．

本論文では，ITRONデバッギングインタフェース仕様の策定目標と，それを達成するための標準化アプローチについて述べる．また，ITRONデバッギングインタフェース仕様が，最も重要な策定目標であるターゲットシステムへの適応性を達成しているかについて評価し，ITRONデバッギングインタフェース仕様の標準化アプローチの有効性を示す．

2. ITRONデバッギングインタフェース仕様の策定目標と標準化アプローチ

この章では，組込みシステムの特徴や開発の現状をふまえ，ITRONデバッギングインタフェース仕様の目的と策定目標，標準化を行うにあたり採用したアプローチについて述べる．

2.1 組込みシステムのデバッグ環境の特徴

組込みシステムのためのソフトウェアデバッグ環境は，次のような特徴を持つ[5),6)]．

- リモートデバッグを基本としている

組込みシステムのソフトウェア開発の大きな特徴として，ソフトウェアの開発に用いるコンピュータ（以下，ホスト）と，開発したソフトウェアを動作させるコンピュータ（以下，ターゲット）が異なる点がある．デバッグ環境も，ホスト上で動作するデバッグツールから，ターゲット上で動作するソフトウェアをデバッグする，いわゆるリモートデバッグを基本としている．ホスト上のデバッグツールがターゲットにアクセスするためには，ICEなどのデバッグ用の機器をターゲットに付けるか，デバッグを支援するソフトウェアをターゲット上で動作させることが必要である．ターゲット上でデバッグを支援するソフトウェアは，独立したソフトウェアとしてターゲットに組み込む方法や，OSにそのような機能を持たせる方法，OS上のタスクとして実現する方法などがあるが，本論文ではそれらをまとめてエージェントと呼ぶ．

- 多様なデバッグ用の機器やソフトウェアが存在する

組込みシステムのソフトウェアのデバッグには，様々な機器やソフトウェアが利用される．プログラムを論理レベルでデバッグする場合には，ソフトウェアデバッガやシミュレータなどが利用されるが，ターゲットシステムを停止できない場合☆☆☆やターゲット上の計算リソースがエージェントを動作させるのに十分でない場合，動作タイミングに依存するバグの発見には，ICEやROMエミュレータなどが利用される．

☆☆☆ 機械を制御しているシステムの場合には，システムを停止させると機械が故障する可能性があるために，停止できない場合がある．

- ホストには十分な計算リソースがある

組込みシステムには，コストダウンの必要性などから厳しいリソース制約が課されおり，ターゲット側の計算リソースにはあまり余裕がないのが通常である．それに比べると，ホストコンピュータは十分な計算リソースを持っている．そのため，デバッグ環境を構築するにあたっては，ホストとターゲット間の通信量や，ターゲット上に必要な計算リソースを減らすことが重視され，ホストの計算リソースがネックになることはほとんどない．

2.2 仕様の目的

ITRON デバッギングインタフェース仕様の目的は，デバッグ環境がμITRON 仕様 OS をサポートするための方法を標準化することである．この標準化により，すべてのデバッグ環境とμITRON 仕様 OSが，標準化された ITRON デバッギングインタフェース仕様に対応することで，任意のデバッグ環境とμITRON 仕様 OS の組合せで，OS サポート機能が利用可能となる．

ITRON デバッギングインタフェース仕様が標準化の対象とする OS サポート機能は，次のとおりである．

- OS オブジェクトの状態表示
- タスクコンテキストの操作
- OS のシステムコールの発行
- OS に依存するブレークポイント機能
- OS の実行履歴の取得

ここで，OS に依存するブレークポイント機能とは，従来のデバッグ環境が持つ指定した番地の実行によるブレーク機能や指定した番地へのアクセスによるブレーク機能に対し，指定したタスクが実行されているなどの，OS に依存する条件を付加する機能を指す．

標準化にあたって，OS サポート機能を持たないデバッグ環境はすでに存在することを前提とし，既存のデバッグ環境に OS サポート機能を追加することに目的を絞った．そのため，デバッグツールとエージェント間の通信手段は既存のものを用いることとし，ここでは標準化項目から除外した．また，ITRON デバッギングインタフェース仕様は標準化を目的としたものであり，この仕様により，タイミングに強く依存するバグに関するデバッグ支援など，これまでになかった新たなデバッグ支援機能が実現可能となるわけではない．

2.3 仕様の策定目標

ITRON デバッギングインタフェース仕様を策定するにあたり，その適用範囲をより広くするために，以下の目標を設定した．

(1) 多様なターゲットシステムへの適用

組込みシステムは，システム規模の面でも特性の面でもきわめて多様である．ITRON デバッギングインタフェース仕様は，多様なターゲットシステムに共通に適用できるインタフェース仕様とすることを目標とした．これを，ターゲットシステムへの適応性と呼ぶ．

(2) 多様なデバッグ環境への適用

タイミングに依存するリアルタイムシステムのデバッグには，ターゲットシステムの動作に影響を与えない ICE や ROM エミュレータなどを用いる必要がある．そこで ITRON デバッギングインタフェース仕様では，ソフトウェアデバッガだけでなく，ICE や ROM エミュレータなどにも共通に適用できるインタフェース仕様とすることを目標とした．

(3) 他の OS やソフトウェア部品への適用

μITRON 仕様に特化する部分を最小限にし，同様のインタフェースで他の OS やソフトウェア部品をサポートするデバッグ環境も構築できることを目標とした．

μITRON 仕様が多様な組込みシステムに適応できる RTOS 仕様であることから，ターゲットシステムへの適応性は，特に重視した策定目標である．とりわけ，計算リソースの制約が厳しいターゲットと，計算リソースに余裕のあるターゲットの，いずれにも有効なインタフェース仕様であることが重要である．

8 ビットや 16 ビットのプロセッサを用いた小規模な組込みシステムでは，ターゲット上の計算リソースの制約が厳しく，エージェントを動作させるために使えるリソースがきわめて小さいのが通常である．そのため，ターゲット上の計算リソースを使わない ICE を用いたり（この意味では，上記の目標 (1) と (2) は関連している），エージェントには最小限の機能のみを実装し，デバッグ機能のなるべく多くの部分をホスト上のデバッグツールで実現する方法が一般的である．

ところがエージェントの機能を最小限にすると，ホスト上のデバッグツールは小さい単位でエージェントを呼び出す必要があり，ホストとターゲット間の通信量は増加する．組込みシステムのデバッグ環境では，ホストとターゲットの間をシリアルインタフェースや Ethernet で接続する場合が多いが，これらの通信路はプロセッサの動作速度と比べると低速であり，通信量の増加はデバッグ効率の低下につながるおそれがある．そこで，ターゲット上の計算リソースに余裕がある場合には，エージェントを高機能化することで通信量を削減し，デバッグ効率を上げる方法が妥当である．

このように，ターゲットシステムの性質によって，ホスト上のデバッグツールとターゲット上のエージェ

ントの最適な役割分担は異なる．デバッギングインタフェース仕様を標準化することで，この役割分担が固定されてしまうと，仕様の適用範囲は狭いものとなってしまう．そこで ITRON デバッギングインタフェースでは，ターゲットシステムに応じてこの役割分担を変更できる，いい換えると，ターゲットシステムへの適応性を持った仕様とすることを目標とした．

2.4 標準化アプローチ

デバッグ環境の OS サポート用インタフェースの標準化には，次の 2 つのアプローチが考えられる．

(1) エージェントに OS サポート機能を実現するための機能を持たせ，その機能の内容や呼び出し方法を標準化する．

(2) OS の内部構造を標準化する．

(1) のアプローチは，既存の多くの OS で用いられてきた方法である．ターゲット上で動作するエージェントに，OS サポート機能を実現するために必要な機能を持たせ，その機能の内容や呼び出し方法などを標準化する．これにより，デバッグ環境は OS の内部構造を知ることなく，OS サポート機能を実現できる．このアプローチはさらに，エージェントの機能を OS 内部に組み込む方法☆と，エージェントを OS 上のタスクとして動作させる方法の 2 つに大別されるが，いずれの方法もターゲットの計算リソースを多く消費するため，リソース制約の厳しいターゲットには適用できるとは限らない．したがってこのアプローチでは，ターゲットシステムへの適応性の目標を達成できない．

(2) のアプローチは，OS の内部構造を標準化することで，どの OS に対しても同じ方法で OS サポート機能を実現できるようにする方法である．たとえば，OS 内のオブジェクト管理ブロックの構造を標準化すれば，デバッグ環境は，OS オブジェクトの状態表示機能を，どの OS に対しても同じ方法で実現することができる．また，システムコールの呼び出し履歴を取得するためには，システムコールの中で定められた番地に必要な情報を書き出すことを標準とすればよい．しかしこのアプローチは，OS の内部構造を規定しないというμITRON 仕様の方針と相反する．OS の内部構造を規定すると，ターゲットとするプロセッサや応用分野に最適化することができなくなり，μITRON 仕様の適用範囲を狭める結果となる．

このアプローチをやや柔軟にした方法として，OS 内のオブジェクト管理ブロックの各フィールドの意味を記述する方法ないしは記述言語を標準化するというアプローチがある．デバッグ環境は，OS メーカが提供した記述を読むことで，OS オブジェクトの状態表示機能を実現することができる．しかし，OS の最適化によりオブジェクト管理ブロックの構造が複雑になるにつれ，構造を記述する言語も複雑化するうえ，他の OS サポート機能に適用しにくいという問題もある．このアプローチによりデバッギングインタフェースを標準化した OSEK/VDX RunTime Interface に関しては，5 章にて述べる．

そこで我々は，OS の内部構造の違いを吸収するモジュールをホスト側に載せるという標準化アプローチを提案した．具体的には，OS サポート機能を実現するためにデバッグ環境が持つべき機能の中で，OS の内部構造に依存する部分を分離し，その部分を実現するモジュールをデバッグ環境に組み込めるようにする．このモジュールを RTOS インタフェースモジュール（以下，RIM）と呼ぶ．また，RIM の機能の内容や呼び出し方法などを標準化する．RIM は OS の内部構造に依存するが，OS メーカが提供すべきとするために問題はない．デバッグ環境は，RIM の機能を用いて OS サポート機能を実現することで，RIM が提供されているすべてのμITRON 仕様 OS に対応可能となる．

このアプローチは，(1) のアプローチでエージェントによって実現していた機能を，ホスト上のモジュールに持たせたものであるといえる．ホスト上のモジュールは，必要に応じて，ターゲット上のエージェントに処理の一部を任せることができ，それによってホストとターゲットの役割分担を変更することが可能になる．

以下では，このアプローチを採用して標準化された ITRON デバッギングインタフェース仕様の概要について述べ，このアプローチによってターゲットシステムへの適応性が実現されていることを確認するための評価を行う．

3. ITRON デバッギングインタフェース仕様の概要

本章では ITRON デバッギングインタフェース仕様[4]のアーキテクチャの概要を述べ，具体例を用いてその動作を説明する．

3.1 構 成

ITRON デバッギングインタフェース仕様に準拠したデバック環境の構成を**図 1** に示す．図中の矢印は既存のインタフェースを，円形と横長の六角形はそれぞれ ITRON デバッギングインタフェース仕様で定める

☆ いい換えると，OS に依存したデバッグ支援機能を実現するための機能を，OS 自身に持たせる方法．ホストとターゲットが共通の汎用 OS では，この方法が一般的である．

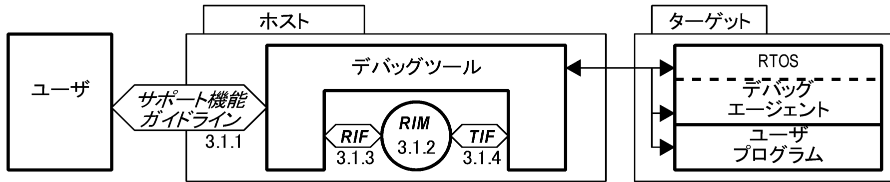

図 1　ITRON デバッギングインタフェース仕様に準拠したデバッグ環境の構成
Fig. 1　Architecture of the debugging environment which supports ITRON Debugging Interface.

モジュールとインタフェースを示す．図中の番号は，本論文中でそれぞれが説明されている節の番号を示す．

この図に示すように，ITRON デバッギングインタフェース仕様は，2 つのインタフェース規約，1 つのガイドライン規約，1 つのモジュールによって構成されている．また実装状況に応じて必要となるその他の規定を含んでいる．

3.1.1　サポート機能ガイドライン

本仕様では，2.2 節で示した OS サポート機能の詳細を，サポート機能ガイドライン☆として定めている．

標準化を行うにあたり，まず既存のデバッグ環境が持つ OS サポート機能を調査し，それを分類・整理して，本仕様でサポートすべき OS サポート機能をサポート機能ガイドラインとして提示した．後述の 2 つのインタフェースは，このガイドラインに示された各機能の実装手法を検討し，それが実現できるように設計を行った．

また，このガイドラインにより，これまでデバッグ環境ごとに統一されていなかった機能の名称や，その機能の振舞いを標準化し，技術者の意思疎通の効率化を図るとともに，デバッグ環境によって利用者がどのような機能を利用できるかを示すための指標を規定している．

3.1.2　RTOS インタフェースモジュール

RTOS インタフェースモジュール（RIM）は，デバッグツールが OS サポート機能を提供する際に必要となる OS 依存の部分を抽出したモジュールであり，デバッグ時に必要となる RTOS に依存した操作を行う．

OS に依存する操作を行うこのモジュールをデバッグツール本体から分離することで，デバッグツールが OS の内部構造に依存しないようにすることができる．またこのモジュールを OS メーカが提供することで，デバッグツールはこのモジュールを組み込むだけで，新しい OS に対応することができる．

RIM と，OS メーカによって提供されるデバッグ用エージェントを総称して，OS 依存モジュールと呼ぶ．

☆ ガイドラインとは，仕様に準拠することが望ましいが必須ではない標準化項目を指す．

表 1　RTOS アクセスインタフェース関数群
Table 1　Functions of RTOS Access Interface.

機能	関数
オブジェクト状態取得	rif_ref_obj（取得）
コンテキスト操作	rif_{get,set}_ctx（取得，設定）
SVC 発行	rif_cal_svc（発行）
	rif_ref_svc（名称-ID 変換）など
ブレーク機能	rif_set_brk（設定）
	rif_ref_cnd（条件式生成）など
トレースログ	rif_{set,get}_log（設定，取得）

3.1.3　RTOS アクセスインタフェース

RTOS アクセスインタフェース（以下，RIF）は，デバッグツールが OS サポート機能を実現する際に用いるインタフェースである．ユーザから OS に依存するような操作の要求があったとき，デバッグツールは RIF を通じて RIM を呼び出す．

RIF は，サポート機能ガイドラインで定めた OS サポート機能を提供するために必要となる，合計 21 個の C 言語 API の関数およびコールバック関数からなる．**表 1** に RIF で提供される主な関数を示す．各関数の詳細に関しては，文献 4) を参照されたい．

3.1.4　ターゲットアクセスインタフェース

ターゲットアクセスインタフェース（以下，TIF）は，RIM がデバッグツールの持つ基本的な操作を行う際に用いるインタフェースである．RIM は RIF を通じて要求された処理を実行するために，TIF を通じてターゲットにアクセスする．

TIF で提供すべき機能は，3.1.1 節で述べた具体的な実装手法の検討の際，デバッグツールで提供すべきとした機能からなる．TIF も C 言語 API の形で定められ，合計 30 個の関数およびコールバック関数で構成される．**表 2** にその主な関数を示す．各関数の詳細に関しては，文献 4) を参照されたい．

3.1.5　その他の規定

以上に加えて，ITRON デバッギングインタフェース仕様には，本仕様をより利用しやすいものとするために，RIM をバイナリ形式で提供する方法を定めた規定や，実行履歴の標準データ形式など，互換性向上

表 2 ターゲットアクセスインタフェース関数群
Table 2 Functions of Target Access Interface.

機能	関数
メモリ操作	tif_{get,set}_mem（読出，書込）
	tif_set_pol（変更通知設定）など
レジスタ操作	tif_{get,set}_reg（読出，書込）
動作操作	tif_sta_tgt（実行開始）
	tif_stp_tgt（強制停止）など
ブレーク機能	tif_set_brk（設定）
	tif_rep_brk（ブレーク通知）など
トレースログ	tif_set_log（設定）
	tif_get_log（情報読出）など

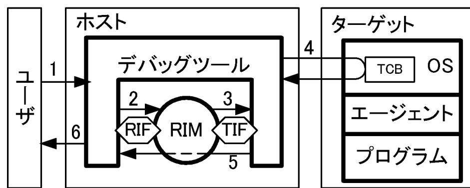

図 2 ITRON デバッギングインタフェース仕様に準拠したデバッグ環境の動作例
Fig. 2 Example of movement of the debugging environment with ITRON Debugging Interface.

を目的とした規定が含まれている．

3.2 動作の具体例

ITRON デバッギングインタフェース上でそれぞれの部分がどのように動作するのか明確にするために，ID=1 のタスクの状態取得の動作を例にあげる．**図 2** 中の番号は，下の各項目番号の処理部分に相当する．

(1) ユーザから ID=1 のタスク状態の取得命令が発行される．
(2) デバッグツールは RIM に ID=1 のタスク状態の取得を要求する．
(3) RIM は対象タスクのタスク制御ブロック（以下，TCB）を読み出すよう，TIF を通じてデバッグツールに要求する．
(4) デバッグツールは，既存の機能を用いて指定されたメモリを読み，RIM に返却する．
(5) RIM は TCB の内容を解析し，標準形式でデバッグツールに返却する．
(6) 得られた結果を基にデバッグツールは ID=1 のタスク状態を表示する，

μITRON 仕様 OS では，タスク状態を取得するシステムコール（ref_tsk）を定めており，このシステムコールを実装している OS では，このシステムコールを用いても同様の操作を行うことができる．その場合は，TCB を読み出すための操作が ref_tsk の実行要求になり，デバッグツールはその実行結果を RIM に返却することになる．

後者の実装方法は関数を外部から呼び出すための機構が必要になる代わりに，TCB を直接読み出す方法に比べカーネル情報の一貫性を保ちやすく，また通信量が少なくなるという特長を持つ．

4. 適応性に関する評価

4.1 評価の目的と方法

2.3 節で述べたように，ITRON デバッギングインタフェースの目標の 1 つにターゲットシステムへの適応性がある．

これは，μITRON 仕様 OS の適用範囲である制約の厳しいターゲットから制約に余裕のあるターゲットまでに ITRON デバッギングインタフェースを適用可能とするためである．

デバッグ環境は制約条件や状況により，多様に変化する．たとえば，リソース制約によって，デバッグ環境の機能には以下のような特性が求められる．

- リソース制約が厳しい場合
  限られたターゲットリソースの下で，所定の機能を実現できること
- リソース制約が緩い場合
  ターゲットリソースの余裕の範囲内で，所定の機能をより高速に動作させること

既存の多くのデバッグツールは，対象となる CPU やユーザの利用方法を想定し，ツールの特性を決めている．しかし本来，これらの特性は OS やユーザの状況によっても変化する．よって，デバッグ環境は必要に応じて，上記 2 つの状況に合わせて特性を変化し，適応することが求められる．

そこで，本仕様の適応性評価を行った．評価の目的は，ITRON デバッギングインタフェース仕様に対応したデバッグツールが，仕様の範囲内で性質の異なる 2 つの環境へ適応できることを確認することである．

評価のために，次のような特徴を持つ 2 つの OS 依存モジュールを用意した．

- 省メモリ実装
  リソース制約が厳しい状況を想定し，限られたリソース範囲内での機能の実現を目的とした実装
- 省通信実装
  リソース制約が緩い状況を想定し，ターゲットに比べ低速であるホスト−ターゲット間の通信路の利用を低減することで，機能の高速動作を目的とした実装

適応性の評価は，ITRON デバッギングインタフェース仕様に準拠したデバッグツールに，上記の 2 つの OS

依存モジュールを導入することで，想定した環境での使用に十分な特性を得られるかをもって評価とする．この評価で重要な点は，デバッグツール本体を修正することなく，OS 依存モジュールの交換のみで，想定した状況への適応が可能となる点である．また適応性の指標として，実行時間，通信データ量，消費メモリの増加量などに着目し，これらを定量的に評価する．

以下 4.3 節と 4.4 節では，ITRON デバッギングインタフェース仕様で提供される機能のうち次の 2 項目に関して行った適応性の評価について述べる．

- タスク ID を条件とするブレークポイント機構
- レディキュー情報の取得

4.2 評価環境

評価環境としてターゲットコンピュータに EPSON 製 CARD-E09A（CPU：日立製 SH7709A[7]，内部クロック：133 MHz，ライトスルーキャッシュ），μITRON 仕様 OS として我々が実装したμITRON4.0 仕様準拠の TOPPERS/JSP カーネルを利用した[8]．ホストとターゲット間の通信はシリアルインタフェース経由（通信速度：115.2 kbps）で行う．

また開発環境として GNU C Compiler を利用し，-O2 で最適化を行っている．デバッグ環境として ITRON デバッギングインタフェース仕様を満たすよう修正を加えた GNU Debugger（以下，gdb）を利用している．

ターゲット上には gdb エージェント（通称スタブ[9]）が組み込まれている．gdb エージェントはメモリ読み出し，書き込み，実行制御など単純なターゲット制御を行うエージェントである．

またターゲットプログラムの実行時間測定は，現実的な最悪値を測定するため，計測開始時点でキャッシュのパージを行っている．

4.3 タスク ID を条件とするブレークポイント機構に関する適応性

デバッグ環境の提供する OS サポート機能の中で最も使用頻度が高いと想定されるのが，タスク ID を条件とするブレークポイント機構（以下，タスク依存ブレーク）である．タスク依存ブレークとは，共有ライブラリなどにブレークポイントを設定した場合にすべてのタスクがその場で停止してしまうことを避け，アプリケーション開発者が注目しているタスクがブレークポイントを通過したときのみ停止するような動作を提供する機構を指す．本節ではこのタスク依存ブレークに着目し，同機構の適応性の評価について述べる．

4.3.1 実　　装

タスク依存ブレークでは，ブレークポイント到達時

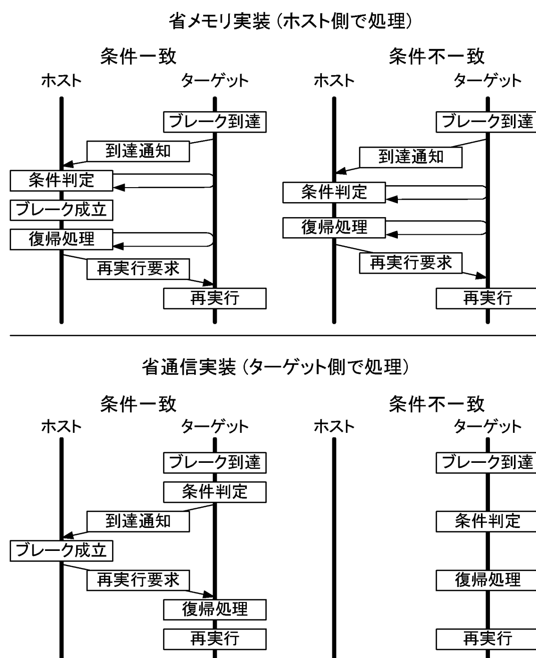

図 3 2 つのブレーク機構の動作
Fig. 3 Behavior of two breakpoint mechanisms.

に現在実行中のタスク ID が条件を満たすか判断する処理を，RIM に実装しホスト上で判断を行うもの（省メモリ実装）と，エージェントに実装しターゲット上で判断を行うもの（省通信実装）の 2 種類を実装した．

今回用いた OS では，内部に実行中のタスクの制御ブロックへのポインタを持つので，このポインタの値が注目すべきタスクの制御ブロックのアドレスと等しくなることをもって条件成立とすることができる．この特性を利用し上記 2 種類のブレーク機構を実装した．

省メモリ実装では，ITRON デバッギングインタフェースのブレーク時コールバック関数（tif_rep_brk）を利用している．tif_rep_brk 関数はプログラムがブレークポイントに達すると呼び出され，返却値によってブレークの成立および不成立の決定を行うことができる．この関数内に条件判定を行うルーチンを実装することで，タスク依存ブレークを実現した．

省通信実装では，OS 上で条件ブレーク機構を実現するための特殊なエージェントを用意した．この条件ブレーク機構は，ブレーク設置アドレス，比較対象アドレス，比較値（32 bit 符号なし整数）をパラメータに持ち，ブレーク設置アドレスの命令実行前に比較対象アドレスの値が条件値と一致した場合にブレーク成立と判断し，gdb エージェントに制御を移す．今回の場合は，現在実行中のタスク制御ブロックへのアドレスを格納している領域を比較対象アドレスとし，注目

したいタスクの制御ブロックのアドレス値を比較値として設定することで，タスク依存ブレークを実現した．

それぞれの判定方式による処理の流れを**図 3** に示す．図中の復帰処理とは，ブレークポイントに置かれた命令を元の命令に置換する処理と再度ブレークポイントを有効にする処理の 2 つを意味している．

評価ではブレーク到達からプログラム再実行までをブレーク処理部とし，これらの処理のうちターゲット上での処理およびホストとの通信処理について調査を行った．なお 4.1 節で述べた理由により，ホストコンピュータ上での処理時間は除外している．

また評価を行うにあたり，タスク数を 8 個，ブレークポイント数を 8 個とした．

4.3.2 結　　果

表 3 に，図 3 で示した処理のうち，ターゲット上のルーチンの実行時間（ホスト上の処理時間，通信にかかる時間を含まず）を示す．省通信実装では，他方と比較し，成立時では条件判定処理のために低速となるが，不成立時では gdb のブレーク時処理が省略されたことにより，高速となった．

表 4 にホストとターゲット間の通信量/パケット量（図 3 の矢印部分）を示す．主な通信内容は到達処理（停止要因，レジスタ内容の通知），実行位置命令の読み出し，再開処理（命令置換，ステップ，ブレーク再設定），再開要求の 5 つである．省通信実装では，再開処理部はターゲットで行うために不要となり，他方よりも通信量が抑えられていることが分かる．

表 5 に実行時間と通信量から算出したターゲット上ブレークルーチンの予想処理時間を示す．予想処理時間は，純粋なターゲット実行時間に，通信処理時間（通信量/通信速度）を加えたものである．この時間は，図 3 のブレーク到達から再実行までから，ホストの処理時間を引いたものに相当する．

表 6 にブレーク機構のためのモジュールを含まないターゲットプログラムと比べて増加したプログラムサイズを示す．省通信実装では，ターゲット上に実装した条件ブレーク機構のため，サイズが増加している．

表 6 より，省メモリ実装では gdb のブレーク機構を利用するため速度の問題はあるものの，既存のターゲットプログラム等にいっさいの変更を加えることなく目的の機能が実装できることが分かる．

一方，省通信実装ではターゲット上で条件判定を行うため，ブレーク不成立時の通信データ量が 0 バイトとなる．それにより省メモリ実装に比べて動作が非常に高速となった．しかしエージェントの導入にともなって 925 バイトのメモリ消費，gdb エージェントに先だってブレーク判定を行うために生じるソフトウェア例外発生時のオーバヘッド（約 3 μs（14%）の増加)，ならびにディスパッチ時にブレークポイントを再設定するために生じるオーバヘッド（約 1 μs（8%）の増加）が発生した．

表 3 ブレーク用エージェントの実行時間
Table 3 Execution times of each break agent.

項目	省メモリ実装	省通信実装
ブレーク成立	44 μs	58 μs
ブレーク不成立	44 μs	38 μs

表 4 ブレーク処理に必要なホストとターゲット間の通信量
Table 4 Amount of communication data with each break agent.

項目	省メモリ実装	省通信実装
ブレーク成立	448 バイト/7	200 バイト/7
ブレーク不成立	448 バイト/7	0 バイト/0

表 5 予想ブレーク処理時間
Table 5 Estimated turnaround time for each breakpoint mechanism.

項目	省メモリ実装	省通信実装
ブレーク成立	32.7 ms	13.9 ms
ブレーク不成立	32.7 ms	38 μs

表 6 ブレーク機構によるターゲットメモリ消費の増加量
Table 6 Increase in target's memory usage of each breakpoint mechanism.

項目	省メモリ実装	省通信実装
コード	0 バイト	788 バイト
データ	0 バイト	137 バイト
合計	0 バイト	925 バイト

上記の結果より，RIM の交換と特殊エージェントの導入のみでデバッグツールに手を加えることなくデバッグ環境の特性を変化させることが可能であることを確認した．

4.4 レディキュー情報の取得に関する適応性

デバッグ環境の OS サポート機能の中でも，オブジェクト状態の表示はタスク依存ブレークに続き利用頻度が高い．そのため ITRON デバッギングインタフェース仕様では同期オブジェクトに関連付けられたタスク ID やその逆，またレディキューやタイマキューに関連する情報を取得する機構を備えている．本節ではその中でも基本的な操作であるキュー走査を含むレディキュー情報の取得に着目し，同機構の適応性の評価について述べる．

4.4.1 実　　装

レディキューの取得とは，現在実行状態（Running）および実行可能状態（Ready）にあるすべてのタスク

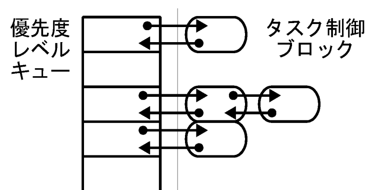

図 4 レディキューの構造
Fig. 4 The structure of the ready queue.

の ID を，優先順位の順に取得することである．

タスク ID は，各タスクの制御ブロックへのポインタから算出した．今回使用した OS では，タスク制御ブロックは固定長の配列として確保されているため，タスク ID は配列のインデックス番号から計算できる．

実行可能状態にあるタスク ID は，レディキューを走査することにより得た．今回用いた OS のレディキューは優先度ごとに存在し，それぞれ該当する優先度を持つ実行可能状態のタスクが連結される構造（外部ハッシュ構造）を用いている．**図 4** に構造を示す．

このレディキュー走査の処理部に関して，4.3 節同様，省メモリ実装と省通信実装の 2 種類の OS 依存モジュールを用意した．

具体的には省メモリ実装では，レディキュー全体の値を取得し，各優先度に対して逐次キューの走査を行う．これらの操作はすべてホスト上でデバッグツールの機能を用いて行われる．メモリ取得動作は（2+実行可能状態にあるタスクの数）回行われる．

省通信実装では，ターゲット上のエージェントによりレディキュー走査処理を実装した．エージェントは例外ハンドラとして登録されており，ホスト上から例外要求を発生し起動する．エージェントは，レディキューからタスク ID のリストを作成する処理をターゲット上で実行し，特定の番地に ID リストとリスト長を連続して格納し，ブレーク状態へ移行する．省通信実装の RIM は，エージェントによって作成されたリストを 2 回のメモリ取得動作によって読み出す（リスト長，リスト本体の計 2 回）．

評価では，ブレーク状態にあるターゲットに対してレディキュー取得処理を行い，これらの処理のうちターゲット上での処理およびホストとの通信処理について調査を行った．なお 4.1 節で述べた理由により，ホストコンピュータ上での処理は除外している．

評価を行うにあたり，優先度の数を 16，タスクの数を 8 としている．また取得時には，1 つのタスクが実行状態であり，7 つの異なる優先度を持つタスクが実行可能状態となっている．

表 7 キュー走査に必要なホストとターゲット間の通信量
Table 7 Amount of communicated data for scanning the ready queue.

省メモリ実装	省通信実装
478 バイト/18	100 バイト/4

表 8 予想キュー走査時間
Table 8 Estimated turnaround time for scanning the ready queue.

項目	省メモリ実装	省通信実装
キュー走査	33.2 ms	7.0 ms

表 9 キュー走査機構によるターゲットメモリ消費の増加量
Table 9 An increase in target's memory usage for scanning the ready queue.

項目	省メモリ実装	省通信実装
コード	0 バイト	164 バイト
データ	0 バイト	200 バイト
合計	0 バイト	364 バイト

4.4.2 結　　果

表 7 にホストとターゲット間の通信量とパケット量を示す．省通信実装では，ホストのメモリ取得動作回数の減少にともない，ホストからターゲットへの命令が減少し，相当する量の通信が減少している．

表 8 に通信量を考慮に入れた予想キュー走査時間を示す．予想キュー走査時間は，ターゲットプログラムの実行時間に，通信処理時間（通信量/通信速度）を加えたものである．ただしこの数値には，ホストコンピュータのオーバヘッド等は含まれていない．双方ともにターゲット上で行う処理の実行時間に比べデータ通信に要する時間が非常に大きいため，プログラム実行時間はデータ転送時間とほぼ等価である．そのため，通信量削減に特化した省通信実装が，減少した通信量に相当する時間分，高速となっている．

表 9 に，エージェント導入にともなうターゲット上のメモリ消費の増加量を示す．省通信実装では，ターゲットに導入したレディキュー走査ルーチンの分，ターゲットメモリを消費している．

これらの表より，先ほどと同様に省メモリ実装は低速ではあるがターゲット上のプログラムを修正することなく実装可能であり，省通信実装は省メモリ実装に比べると多少オーバヘッドをともなうものの，通信データ量が少なくなることで高速に処理を行うことが可能であることが分かる．以上より，4.4.2 項と同じように，ITRON デバッギングインタフェース仕様のレディキュー情報の取得機能に対する適応性を示すことができた．

```
IMPLEMENTATION_ORTI_2_0 {
    INT8 PRIORITY,"優先度";
}
TASK 1 {
    PRIORITY = "osTaskCB[1].task->priority";
};
```

図 5 ORTI ファイルのサンプル
Fig. 5 Example of ORTI file.

5. 関連研究

本章では，ITRON デバッギングインタフェース仕様に類似する標準仕様を取り上げ，それぞれの技法と問題点等を述べる．

- OSEK/VDX RunTime Interface（ORTI）

ORTI [10]は，OS の各オブジェクトの制御ブロックの構造を記述する方法を定め，デバッグツールはその記述を基に情報の取得を行うことで，OS サポート機能の実現を可能とする．**図 5** に，タスク優先度を例として，ORTI ファイルに記述内容を示す．この手法は 2.4 節の (2) に相当し，同節で示したような問題を持っている．

- Java Platform Debugger Architecture（JPDA）

JPDA [11]は，JavaVM 上で動作するプログラムをデバッグするための仕様である．この仕様では，JavaVM 上のデバッグ支援エージェント，デバッグツールと JavaVM 間の通信手法，およびデバッグツールがこれらの機能を利用するための Java インタフェースを定めている．これにより，μITRON 仕様と同じように，Java という共通の仕様を基に独立して実装された JavaVM に依存せず，高い移植性を持ったデバッグツールの作成を可能とする．しかし JPDA では，デバッグツールとエージェント間のプロトコルを定めているため，その間で負荷の調整ができず，組込みシステム用のデバッグ環境に求められている適応性が実現できない．

- Nexus 5001

Nexus5001 [12]は，プロセッサごとに変化するデバッグツールを統一するための仕様である．ターゲットプロセッサは，Target Abstraction Layer という RIM に似た部分をデバッグツール上に設けることで，既存の JTAG インタフェースを利用して任意の Nexus 対応デバッグツールでデバッグを行うことが可能となる．またこの仕様では，より高度な機能を提供するために，JTAG を拡張した独自の通信インタフェースもある．Nexus は OS サポート機能が対象ではないが，ホスト上のモジュールでターゲットの違いを隠蔽するという点で，我々のアプローチと類似している．

6. まとめ

本論文では，μITRON のような仕様は同一であっても実装が異なるような RTOS に対するデバッギングインタフェースの標準化アプローチを提案し，それを適用して標準化した ITRON デバッギングインタフェース仕様について述べた．

また ITRON デバッギングインタフェース仕様の目標の 1 つである適応性について評価し，目標の達成を確認するとともに ITRON デバッギングインタフェース仕様の有効性を示した．ITRON デバッギングインタフェース仕様の特長は，RIM とエージェトという 2 つのモジュールを組み合わせることで，デバッグツールに依存することなく同一の枠組みの中でデバッグ環境の特性を変化できるという点である．この特徴は，RTOS およびプロセッサの特性や，ユーザの利用状況など様々な状況によってデバッグ環境に求められる特性が変化するような組込みシステムの分野などでは有用である．

今後は ITRON デバッギングインタフェース仕様を ICE に適用し，OS 依存モジュールがどの程度共通化できるかを調べることで，2.3 節の目標 (2) の達成に関する評価を行うとともに，本手法がタイミングに依存するリアルタイムシステムのデバッグにも有効であるかについても評価する予定である．

現在，ITRON デバッギングインタフェース仕様はインタフェースがμITRON 仕様に依存しているが，アーキテクチャ自身は他の OS やミドルウェアなどの任意のモジュールに適用できると考えている．そこで，今後の課題として，本インタフェースの汎用性を高めるため，オブジェクトの種類や仕様に依存した情報を記述する手法などを検討し，インタフェースの実装がそれらに依存することを避けるようにする予定である．

謝辞 ITRON デバッギングインタフェース仕様は社団法人トロン協会 ITRON 部会 ITRON デバッギングインタフェースワーキンググループの成果である．この場をお借りして仕様の標準化に向けて尽力してくださった関係者の方々に深く感謝する．

参考文献

1) 坂村 健（監修），高田広章（編）：μITRON4.0 仕様 Ver. 4.00.00，トロン協会 (1999).
2) 高田広章，田丸喜一郎：ITRON サブプロジェクト第 2 フェーズへの展開，情報処理，Vol.40, No.3, pp.223–228 (1999).

3) 株式会社ワイ・ディ・シー：microVIEW-G ユーザーズマニュアル（Ver. 3.1a），9th edition，株式会社ワイ・ディ・シー advice 事業部 (2000).
4) トロン協会 ITRON 部会：ITRON デバッギングインターフェース仕様 ver 1.A0.00，トロン協会 (2000).
5) Barr, M.: *Programming Embedded Systems in C and C++*, O'REILLY (1999). 有馬三郎（訳）: C/C++による組み込みシステムプログラミング，オライリージャパン (2000).
6) 宿口雅弘：組込みシステムのデバッグ技法，情報処理，Vol.38, No.10, pp.886–891 (1997).
7) 日立製作所：SH7709 ハードウェアマニュアル，4th edition，日立製作所半導体グループ電子統括営業本部 (1998).
8) TOPPERS/JSP プロジェクトホームページ．http://www.ertl.ics.tut.ac.jp/TOPPERS/.
9) Stallman, R.M. and Pesch, R.H.: *Debugging with GDB fifth edition*, Free Software Foundation (1998). コスモプラネット（訳）：GDB デバッギング入門，アスキー出版 (1999).
10) OSEK/VDX: *OSEK/VDX Runtime Interface 2.0* (*Draft H*) (2001).
11) Sun Microsystems Inc.: Java Platform Debugger Architecture (2000). http://java.sun.com/products/jpda/.
12) Forum, T.N.: IEEE-ISTO 5001-1999, The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface (1999). http://www.ieee-isto.org/Nexus5001/.

(平成 12 年 12 月 18 日受付)
(平成 13 年 4 月 6 日採録)

若林　隆行

2001 年豊橋技術科学大学大学院情報工学専攻修士課程修了．現在，同大学院電子・情報工学専攻に在学．リアルタイム OS，デバッグ技法，ソフトウェア開発環境の研究に従事．修士（工学）．

高田　広章（正会員）

豊橋技術科学大学情報工学系講師．1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同学科の助手などを経て，1997 年 12 月より現職．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術などの研究に従事．ITRON 仕様の標準化活動に，中心的メンバとして参加．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．